Cyclist: Accelerating Hardware Development

Jonathan Bachrach, Albert Magyar, Palmer Dabbelt, Patrick Li, Richard Lin, Krste Asanovic,
Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
Email: { jrb, magyar, palmer.dabbelt, psli, rlin, krste } @berkeley.edu

***Abstract*—The end of Dennard scaling has led to an increase in demand for energy-efficient custom hardware accelerators, but current hardware design is slow and laborious, partly because each iteration of the compile-run-debug cycle can take hours or even days with existing simulation and emulation platforms. Cyclist is a new emulation platform designed specifically to shorten the total compile-run-debug cycle. The Cyclist toolflow converts a Chisel RTL design to a parallel dataflow graph, which is then mapped to the Cyclist hardware architecture, consisting of a tiled array of custom parallel emulation engines. Cyclist provides cycle-accurate/bit-accurate RTL emulation at speeds approaching FPGA emulation, but with compile time closer to software simulation. Cyclist provides full visibility and debuggability of the hardware design, including moving forwards and backwards in simulation time while searching for trigger events. The snapshot facility used for debugging is also used to provide a "pay-as-you-go" mapping strategy, which allows emulation to begin execution with a low-effort placement, while higher-quality emulation placements are optimized in the background and swapped in to a running emulation. The Cyclist ASIC design requires $0.069mm^2$ per tile and runs at 2GHz in a 45nm CMOS process. Our evaluation demonstrate that Cyclist outperforms FPGA emulation, VCS, and C++ simulation on combined compile and run time for up to a billion cycles for a set of real-world hardware benchmarks.**

***Index Terms*—Simulation; RTL; Design; FPGA; Hardware; Modeling; Debugging**

I. Introduction

The end of traditional Dennard scaling is driving demand for efficient specialized hardware accelerators in all scales of computing system from smartphones [1] to datacenters [2]. But designing efficient digital hardware is incredibly labor intensive compared to sequential software of similar functional complexity, as hardware designers must not only verify highly concurrent implementations but also optimize performance and energy consumption. The typical design loop contains many steps that take considerable wall-clock time with existing hardware design tools. Each iteration of the design loop requires programming, compilation, testing, debugging, and evaluation steps. To accelerate development, we need to decrease the total time through the many required iterations of the design loop.

In general, reducing the latency of the steps of the loop where the designer is not productively developing (compilation and emulation) will speed the development process. To this end, hardware designers typically use a variety of emulation techniques at different stages in the design process, trading off cost, compilation speed, and emulation throughput. Emulation techniques range from software emulation on general-purpose servers to hardware emulation on specialized multi-million dollar hardware emulation engines. We use the term "target" to describe the design being emulated, and "host" to represent the system emulating the target design as shown in Figure 2. Although software-based emulations are sometimes called simulations, in this paper we use the term emulation for any cycle-accurate and bit-accurate model of a target, whether implemented in software or in hardware.

Figure 1 shows the summed compile-and-run time curves for software and FPGA-based emulation alternatives. Software-based approaches, either using compiled C++ models or Verilog models compiled by Synopsys VCS [3], have fast compile times but relatively slow run times, and work well for fewer emulated cycles. FPGA-based emulation has very slow compile times, due to the FPGA

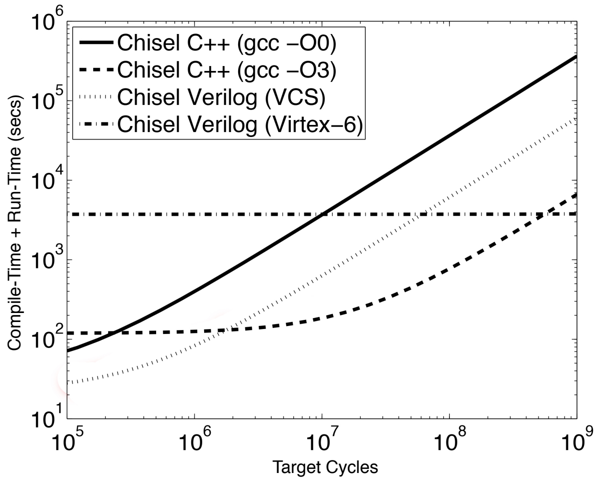


Fig. 1: Example "time-to-cycle-N" plot for an in-order single-issue processor booting an simple OS, where "time-to-cycle-N" = compile-time + run-time. Various emulation techniques are shown including GCC-compiled C++ code, a Verilog simulator (VCS), and an FPGA system (Virtex-6). Lower times are better. For the FPGA, initial high compilation times are eventually amortized over many target cycles.

synthesis and place-and-route tools, but has very fast run times and works better for situations involving many emulated cycles.

Early in development, designs often fail after a few target cycles due to simple errors. After the simpler bugs are fixed, more subtle and complex bugs tend to only manifest in tests running for much larger numbers of target cycles. Hence, for early development, users prioritize emulators with good debugging capabilities, increased visibility of internal state, and fast compilation times, and are less concerned with emulation speed. Later in development, users must prioritize fast emulation speed, and are forced to accept much longer compilation times and less visibility. This makes it exceptionally tedious to find and debug errors that occur only after a large number of cycles, as fast emulation platforms typically require a time-consuming re-synthesis to trace different sets of signals.

Designers try to select emulation strategies to be on the lowest curve for the expected number of target execution cycles. Unfortunately, there is considerable effort required to set up all the various emulation solutions, and it is also difficult to predict how many emulation cycles will be needed to reproduce a bug. Ideally, a developer would use just one emulation solution and would incur total turnaround time proportional to the number of emulation cycles actually required, and would also be able to have full visibility into the design even at high emulation speed.

This paper introduces Cyclist, a new hardware emulation system that accelerates the design loop by providing fast emulation speed, pay-as-you-go compilation time, and full interactive visibility into the target design. Cyclist employs a parallel emulator architecture consisting of a spatial array of specialized processing tiles, onto which RTL designs are mapped using the Cyclist toolflow, as shown in Figure 9.

Compared to a software emulation approach, Cyclist has slightly

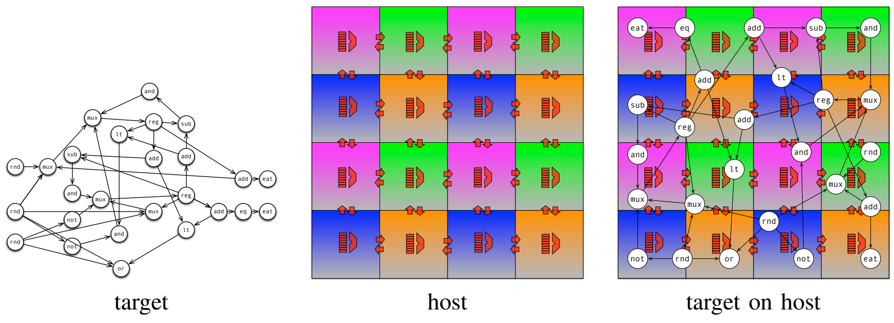


Fig. 2: Target circuit, emulator host machine, and target circuit being emulated.

longer compilation time, but much faster execution due to the large number of parallel elements and network-on-chip specialized for fine-grained synchronization common in RTL simulation.

Compared to an FPGA emulation approach, Cyclist increases emulation capacity by representing the circuit graph using coarse-grain instructions stored in the processing tile's memory, rather than as a logic circuit mapped to programmable logic blocks. Compilation time is also much reduced as only a coarse-grain dataflow graph of RTL operators needs to be placed and scheduled, rather than placing and routing a gate-level circuit netlist. By using simple pipelined processors, host clock rate can be much greater than for placed FPGA design, which helps improve efficiency. Cyclist also makes use of the software-programmable tiles to provide high visibility, with support for interactive testing, debugging and evaluation.

Cyclist's benefits are that it is:

- **efficient to compile to:** 10–100× faster than FPGA,
- **efficient to run:¡** 1–10× slower than FPGAs,
- **quick to probe any signal:** no recompile necessary, and
- **easy to map large designs:** high capacity plus graceful slow-down with size.

In the rest of the paper, we first compare Cyclist against earlier related work, then describe the hardware architecture and software toolflow. We additionally present a user-friendly interactive debugging platform. Finally, the system is evaluated using a range of large, real-world RTL designs.

II. Related Work

The cheapest and lowest-performance form of emulation uses general-purpose computers running emulation code. Newer software emulators compile RTL into efficient C++ code that implements the exact function and timing semantics of the design [4]. This approach provides fast compile times, full visibility, and can handle large designs, limited only by host memory capacity. The main disadvantage is emululation speed, with target emulations tending to run in the KHz range, creating onerous wait periods for long simulations on large designs.

Another common emulation approach is map the RTL design to FPGA boards [5], [6]. The primary advantages are that the FPGA boards can run a design with target clock rates around the 10–50MHz rate, and the FPGAs boards represent a modest additional expense. Unfortunately, mapping designs intended for a custom ASIC to an FPGA is challenging, as the FPGAs available in a given generation have much lower capacity than ASICs in the same technology generation. Heroic emulation design efforts are necessary when designs are bigger than what will fit in a single FPGA [7] [8], and often, multi-FPGA emulations run much slower, dropping to the sub-1 MHz range due to chip-chip communication. Partitioning across multiple FPGAs is usually done manually, although tools are now available for automatic partitioning [9]. One of the greatest challenges in using FPGAs for emulation, is that visibility is poor. Often the FPGA emulation must be re-synthesized to observe a different set of internal signals.

Some of the earliest work in building specialized engines for hardware emulation were the IBM Yorktown Simulation Engine [10] (YSE) and the IBM Engineering Verification Engine [11] (EVE). YSE and EVE are essentially simple 4-bit processors that are connected by a crossbar network. The modern-day Palladium machines are an evolution of these earlier ideas [6], and can handle up to 2 billion gates, yield up to 4MHz target frequency, and support up to 512 users. The tool flow can compile up to 35M gates / hour on a single PC, provide full visibility to all signals with little slowdown, and integrates with logic and power simulation, SystemC, and prototype hardware. The main downside to the Palladium machines are their high cost, being priced in the several million dollars range due to the low effective capacity of the design.

Cyclist uses a similar network-of-specialized-processors approach as the dedicated simulation engines, but maps designs at the RTL-operator level instead of the gate-level to increase capacity, and uses a mesh network rather than a crossbar to provide better scaling to larger numbers of emulation engines. The Cyclist toolflow is responsible for scheduling all the emulation traffic across the inter-tile network.

Malibu [12] is a recent emulation system that is again a statically scheduled network of processors. Malibu differs from Cyclist in having a fine-grained FPGA subsystem on each processor and a wide-word VLIW instruction format. Cyclist is more dynamically scheduled than Malibu, supporting interlocked dataflow execution with less stalling, and also provides novel debugging support for triggering, tracing, snapshotting, and stepping.

In summary, Table I shows various simulation options with which to compare Cyclist. In short, Cyclist is a low cost emulation design which makes it easy to map designs, has fast compilation and run speeds, allows visibility at speed without recompilation and in circuit.

III. Cyclist Architecture

In this section, we describe the architecture of the Cyclist hardware platform. A Cyclist machine consists of a number of computational elements, known as "tiles", connected by an on-chip network. The current Cyclist implementation uses five-stage RISC-like processors as tiles, which are connected by a statically-scheduled mesh network. Each tile's network ports are register-mapped and interlocked.

name	ease of mapping	compiler speed	run speed	visibility speed	visibility recompile?	in circuit?	cost
Verilog	easy	fast	slow	slow	no	no	low
C++	easy	fast	med	slow	yes	no	free
FPGA	hard	slow	fast	fast	yes	yes	med
Palladium	easy	fast	fast	fast	no	yes	high
Cyclist	**easy**	**fast**	**fast**	**fast**	**no**	**yes**	**med**

TABLE I: Comparison of various simulation options according to a number of important simulation attributes.

op	**dst**	**x**	**iy**	**y**	**z**	**in**	**out**
5b	*4b*	*5b*	*1b*	*5b*	*5b*	*2b*	*4b*

TABLE II: Cyclist RISC-style instruction format including one destination specifier, three operand specifiers `x`, `y`, and `z`, and one network route specified by `in` and `out`. The `y` field holds an immediate when `iy` is one. For certain instructions, `z` is an immediate specifying the number of mask bits with which to mask the result.

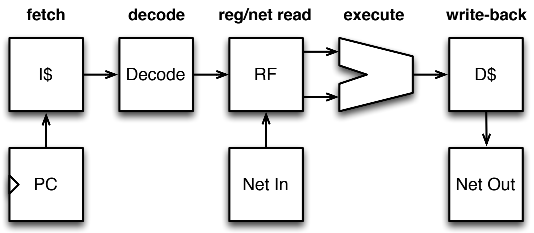


Fig. 3: Cyclist five stage pipeline.

A. The Cyclist Pipeline

Each Cyclist tile consists of a 32-bit RISC-like pipeline that is tightly coupled with the on-chip network. Cyclist uses a proprietary RISC ISA, as shown in Table II. There are 32 architectural registers, and emulation-specific features such as bitwidth masking of all instructions and direct network read and write are included to enhance performance on common operations. In a significant departure from general-purpose chips, there are no control flow instructions, as they are not used by the scheduled logic emulation code. Some additional emulation-specific instructions are `rst` for reset, `mux` for conditionals, `log2` for find highest bit, and `cat` for concatenate two fields. A diagram of the Cyclist pipeline is shown in Figure 3.

Each tile has access to two large on-chip SRAMs: the code memory and the data memory. The code memory stores the instructions that each Cyclist tile executes. Each instruction is 32 bits wide and consists of an ALU operation in addition to a network operation (either of which can be NOPs). The data memory provides storage for target design state as well as register spills. These two memories determine the size of design that can be emulated on a Cyclist chip: larger memories allow for larger designs to be emulated at the cost of emulation speed, as area that could be used for ALUs is taken by SRAMs.

In the presented Cyclist implementation, both the code and data memories are 1024 words long (32kbit). These sizes were chosen in order to hit a target clock of 1MHz when fully utilized; however, capacity can be increased at the cost of lower simulation speed when fully utilized and a slower critical path.

The biggest difference between a Cyclist tile and a standard RISC processor is that Cyclist tiles have no control-flow instructions. Aside from the implicit loop around all instructions execution is sequential, and aside from interlocking on the network ports there are no stalls. This allows for an extremely simple pipeline, which is key to achieving a high clock rate while keeping area small.

op	**d**	**x**	**y**	**z**
special				
`nop`	_	_	_	_
inputs				
`rst`	*d*	_	_	_
`lit`	*d*	*val*		
logical				
`not`	*d*	*x*	_	*w*
`and`	*d*	*x*	*y*	_
`or`	*d*	*x*	*y*	_
`xor`	*d*	*x*	*y*	_
`eq`	*d*	*x*	*y*	_
`neq`	*d*	*x*	*y*	_
`mux`	*d*	*c*	*t*	*e*
`log2`	*d*	*x*	_	*w*

bits				
`lsh`	*d*	*x*	*n*	_
`rsh`	*d*	*x*	*n*	*w*
`rsha`	*d*	*x*	*n*	*w*
`cat`	*d*	*x*	*y*	*wy*
arithmetic				
`add`	*d*	*x*	*y*	*w*
`sub`	*d*	*x*	*y*	*w*
`lt`	*d*	*x*	*y*	*w*
`gte`	*d*	*x*	*y*	*w*
`mul`	*d*	*x*	*y*	*w*
state				
`ld`	*d*	*a*	*e*	_
`st`	*a*	*x*	*a*	*e*
`ldi`	*d*	*a*		
`sti`	*a*	*x*	*a*	

TABLE III: 24 instructions necessary for implementing Cyclist RTL functionality.

B. The Cyclist Network

Cyclist's on-chip network is responsible for feeding each tile with data for execution, and it is therefore a first-order performance constraint. Th network was designed to be low latency, handle fanout efficiently, have a simple hardware implementation, and be easy to schedule for. A block diagram of Cyclist's on-chip network is shown in Figure 4.

The on-chip network consists of an interlocked, NSEW-mesh network. Each compass direction consists of a pair of one element blocking queues with one used for input and one used for output. In order to avoid physical overhead for routing, tiles are responsible for moving data from inputs to outputs; routing across the chip is done statically at design compilation time. Each instruction can read from up to one network input port, compute, and then broadcast to any number of network output ports (specified using the `out` field and shown in Table II), where the network output value can be directly from the network input or from the result of the compute. Any arithmetic instruction may use a network port as any of its input or output operands. Instructions specifying the network destination register utilize the network output bitmask, which permits multicast of their writeback values to any subset of adjacent tiles. If the network ports are unused for a given arithmetic instruction, a network operation may be performed in parallel, allowing a higer degree of instruction-level parallelism. Careful scheduling of network traffic allows for efficient handling of fan-out to multiple tiles.

The pipeline is interlocked and will stall if space on an output queue is full or an input queue has no data. This allows the compiler to avoid emitting spurious NOPs while still maintaining a fully static schedule. This interlocking affords a significant increase in code density without which Cyclist would not be a practical machine.

Cyclist includes a host interface on each tile and connected as a debug scanchain to the host as shown in Figure 4. The interface

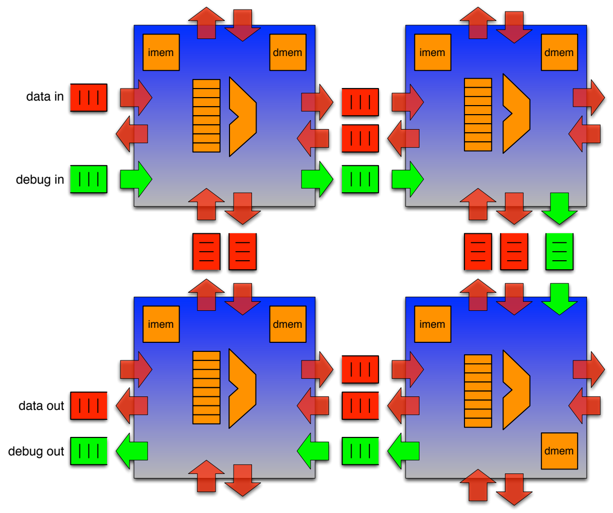


Fig. 4: Tiled emulation machine with statically scheduled mesh network (shown in red) and debug scanchain (shown in green).

supports peek and poke commands for reading and writing host/target architectural state, and a step command for stepping the target machine n target cycles. Tiles can be individually addressed or broadcast to. Every instruction includes a trace bit, that when set, sends out the instruction result out the debug scanchain.

C. ASIC and Performance Results

We have pushed our Cyclist design through Synopsis DC and ICC [1] Placed-and-routed results using models from a 45nm CMOS process produced favorable results.

tile area	=	0.069 mm^2 / tile
array speed	=	1.97GHz

Given this area, we could, for example, fit 280 tiles and a RISC-V [13] Rocket host CPU on a 3mmx6mm die in a 45nm process. The area of a given tile breaks down as shown in Figure 5. Approximately 60% of the area is memory, 16% is register files, and finally 9.2% is debug interface area. Finally, the static power reported from the Synopsis tools is 16.8mW per tile.

A number of Chisel benchmark circuits were synthesized for a Cyclist as listed in Table IV. Cyclist is able to achieve MHz target cycle frequencies as shown in Figure 6. Figure 7 shows the results of a comparison of Cyclist to Chisel generated C++ simulation performance and Figure 8 shows the simulation performance relative to Xilinx Kintex-7 FPGA[2] using Xilinx Vivado 2013.4 tools[3]. Cyclist is approximately 11.0x faster than C++ and is 16.8x slower than an FPGA[4].

IV. Cyclist Workflow

Despite the fact that Cyclist was designed to be a target that is easy to map to, most of the implementation complexity lies in the toolchain used to map designs for execution. Cyclist currently

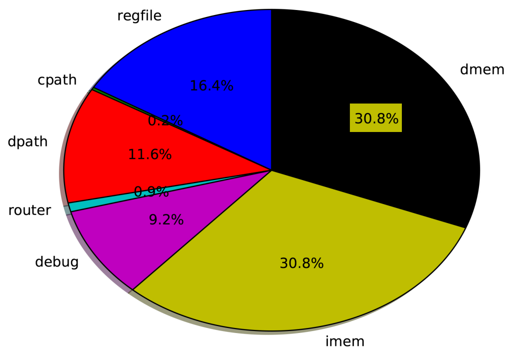


Fig. 5: Area breakdown for single Cyclist tile. Tile area = 0.069 mm^2 in a major 45nm CMOS process.

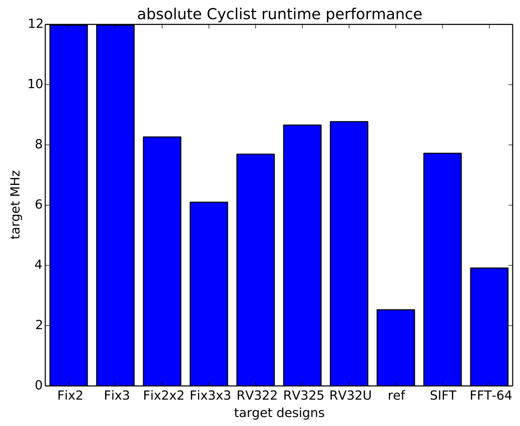


Fig. 6: Absolute run time performance of various benchmarks on Cyclist in MHz range. Cyclist achieves desired MHz target cycle frequency with an overall average of 7.8 MHz target cycle frequency.

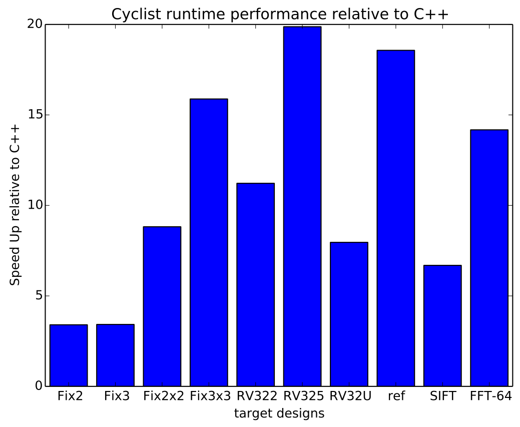


Fig. 7: Relative run time performance of various benchmarks on Cyclist compared to Chisel generated C++. Cyclist is on average 11.0x faster to execute than C++.

[1]The only outstanding issues with our ICC results are related to closing timing with external delays in off-chip connections. Future work will include more realistic chip-level IO.

[2]XC7K325TFFG900-2

[3]We were only able to synthesize a subset of the benchmarks on our FPGA.

[4]Comparing Cyclist fabbed with 28nm technology against an FPGA fabbed using 28nm technology is a slightly unfair comparison.

type	Cyclist				RISC-V				DSP	
kind	Single Tile		Array		Sodor			Rocket		
name	*Fix2*	*Fix3*	*Fix2x2*	*Fix3x3*	*RV322*	*RV325*	*RV32U*	*RefChip*	*SIFT*	*FFT-64*
size	654	665	2475	6140	1202	1470	1127	12579	1554	8449

TABLE IV: Series of benchmarks with given sizes in number of RTL graph nodes. Fix2 and Fix3 are two stage and three stage pipeline implementations of Cyclist. RV322, RV325, and RV32U are educational RISC-V processors with two stage, five stage, and microcode microarchitectures. Refchip is an in order RISC-V processor, called Rocket, with non blocking cache based uncore. N.B., RefChip's verification on Cyclist is currently in progress. SIFT is a low level SIFT feature extraction circuit. FFT-64 is a direct 64 input 32 bit FFT. The number of gates is often 10-20x number of RTL graph nodes.

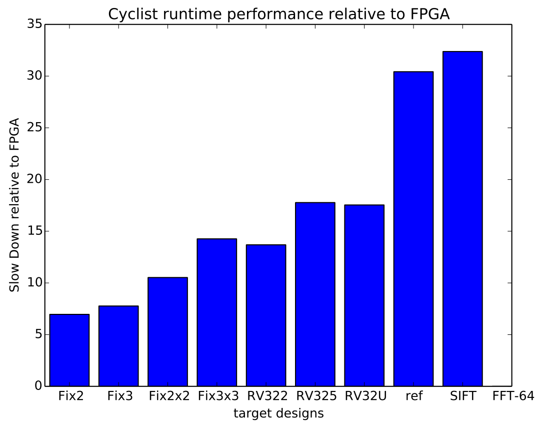


Fig. 8: Relative run time performance of various benchmarks on Cyclist compared to FPGA (Xilinx 28nm Kintex-7). Cyclist is on average 16.8x slower to execute than an FPGA.

assumes designs are written in Chisel [4] [5], UC Berkeley's powerful hardware construction language that supports multiple backends from the same source description. The Cyclist toolchain is implemented in phases that are somewhat similar to an FPGA toolchain (shown in Figure9): first, high-level Chisel designs are mapped to Cyclist instructions, then these instructions are placed on tiles in the fabric, and finally each instruction is scheduled for execution.

The mapping phase is the simplest: during circuit elaboration Chisel builds a graph that contains every node and operation necessary to synthesize the circuit. Each one of these nodes maps directly to a set of Cyclist instructions, so all that is required is to emit the corresponding code.

Instructions are placed on tiles by using simulated annealing. The cost function is the sum of the Manhattan distance between every node in the graph, which approximates the communication cost when executed on the network. This cost function has the advantage of being linear, which allows us to quickly compute the cost of a perturbed graph by only calculating the cost difference caused by the perturbation. This allows for a fast parallel variant of simulated annealing [14] to be used that should scale well. Finally, nodes are only moved if they result in an admissible layout that doesn't oversubscribe any tile resources. This means, that layouts are always admissible during the entire annealing process.

Instructions are scheduled for execution by simulating an execution of the computation graph on a virtual Cyclist-like machine, recording a trace of the execution including network routing, and emitting that trace on the statically scheduled machine. This allows the scheduler to be fairly agnostic of actual machine details at the cost of only allowing for a greedy schedule to be performed. As the toolchain is designed to execute quickly, it seems a greedy algorithm is probably a sane approach so we just stuck with it.

[5] The conceptual input to Cyclist is actually RTL, and thus, in principal any synthesizable RTL description is runnable on Cyclist.

Target registers and memories are mapped to host memory. During scheduling, host registers are allocated and potentially spilled to remaining host memory. The final schedule is split into two phases: a **combinational phase** for computing outputs and next state values, and a **state update phase** for committing next state such as target registers and memories.

Routing is optimized to minimize fanout routing. Instead of sending out fanned out data one by one to multiple destinations, during scheduling, fanout data is arranged step by step according to local ports traversed, where paths sharing local ports are coalesced. The result is a spanning tree of the fanout paths with only $O(log_n)$ of the amount of routing work required.

After producing a static schedule, a peephole optimizer removes unnecessary NOPs in order to compress the code for Cyclist. As an additional benefit, Cyclist is able to execute code in a spatially pipelined fashion.

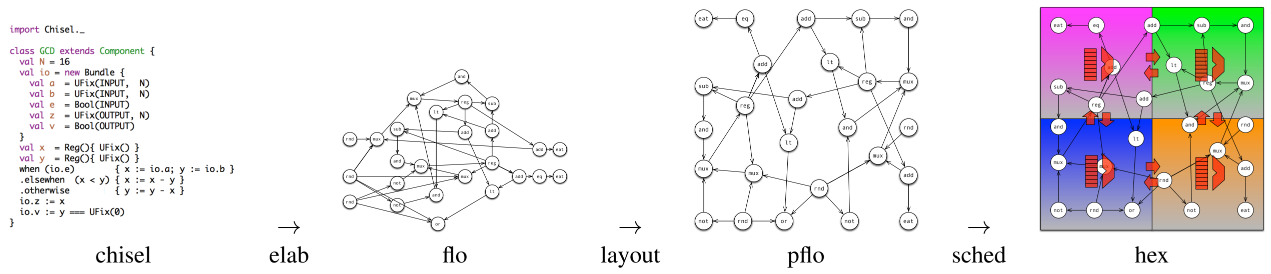
Figure 10 shows the compilation time for the various benchmarks shown in Figure IV. Figure 11 shows the results of a comparison of Cyclist to C++ compiler speed on Chisel generated C++ and Figure 8 shows the Cyclist's compiler performance using Xilinx Vivado 2013.4 tools while targeting Xilinx Kintex-7 FPGA. Cyclist achieves an average 7.7x slow down over C++ and an average 3.0x speed up over an FPGA.

A. *"Pay as You Go" Compilation*

Cyclist has been designed to make it easy to compile for, and, as a result, its runtime performance improves with increased compilation time as shown in Figure 13. From this we can also see that our compiler performance numbers shown in Figures 10, 11, and 12 are conservative because the compiler is able to hit the peak runtime performance with about $1/3$ the annealing steps allotted. With the addition of fast snapshot support, we can update the layout periodically and gradually improve runtime performance. We do this by running the compiler in parallel and periodically loading a new layout.

We update a new layout by saving out a snapshot, loading a new layout, and then restoring saved snapshot into new layout. Snapshots contain the entire circuit state. Snapshots are run at a regular interval as shown in Figure 14. Figure 15 shows that by running compiler in parallel and periodically updating the layout, Cyclist is faster than all emulation techniques until about a billion cycles.

Cyclist holds all snapshot state in host memory. State can be ordered once to give offsets to each piece of state. A save snapshot command is just a sequence of peek packets, and similarly, a restore snapshot command is a sequence of poke packets with stored state as values. Save and restore snapshot commands can be prepared ahead of time and sent out in order in pipeline fashion one packet per cycle.



```
import Chisel._

class GCD extends Component {
  val N = 16
  val io = new Bundle {
    val a  = UFix(INPUT,  N)
    val b  = UFix(INPUT,  N)
    val e  = Bool(INPUT)
    val z  = UFix(OUTPUT, N)
    val v  = Bool(OUTPUT)
  }
  val x  = Reg(){ UFix() }
  val y  = Reg(){ UFix() }
  when (io.e)         { x := io.a; y := io.b }
  .elsewhen  (x < y) { x := x - y }
  .otherwise          { y := y - x }
  io.z := x
  io.v := y === UFix(0)
}
```

Fig. 9: Compilation workflow for Cyclist involving elaboration, layout, and scheduling phases.

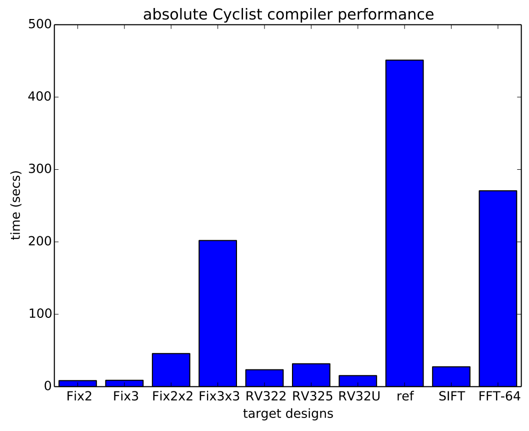


Fig. 10: Absolute compile time performance of various benchmarks on Cyclist in seconds range.

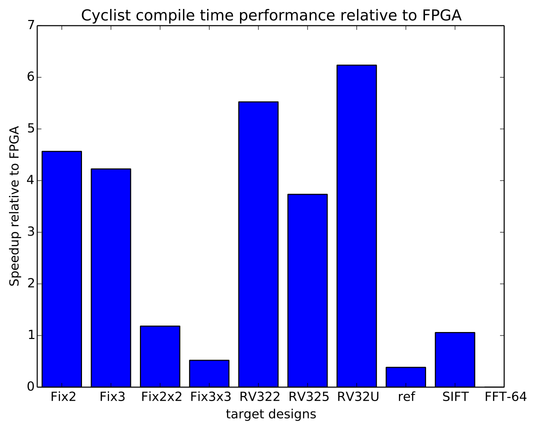


Fig. 12: Relative compile time performance of various benchmarks on Cyclist compared to FPGA (Xilinx 28nm Kintex-7) using Vivado. Cyclist is on average 3.0x faster to compile for than an FPGA.

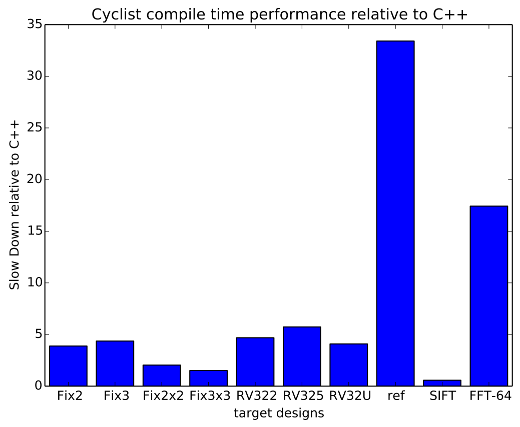


Fig. 11: Relative compile time performance of various benchmarks on Cyclist compared to Chisel generated C++ in g++-4.8 using -O3. Cyclist is on average 7.7x slower to compile than C++.

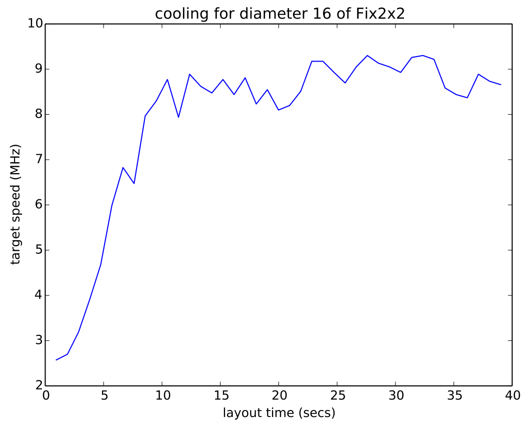


Fig. 13: Cyclist runtime performance improves smoothly over compilation time for an example of a 2x2 Cyclist array target circuit being emulated. Runtime performance starts out quickly at approximately 2.5 MHz at 1 second and then plateaus at approximately 9 MHz in approximately 10 seconds.

V. Interactive Visibility Debugging

Full visibility is a highly sought-after debugging feature for emulators which allows users to query the value of any internal circuit wire. Even after extensive unit testing of hardware components in isolation, it is still common for components to interact incorrectly. Frequently in these scenarios, the easiest method for debugging the circuit is to observe the internal signals sent among the components. Furthermore, designers do not typically know *which* internal wires they need to observe in order to find the inconsistency. The ultimate

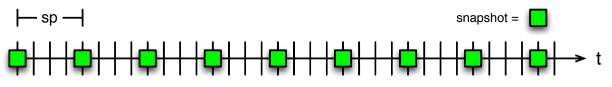


Fig. 14: Periodic snapshots and snapshot interval between snapshots, where `sp` is the Snapshot Period.

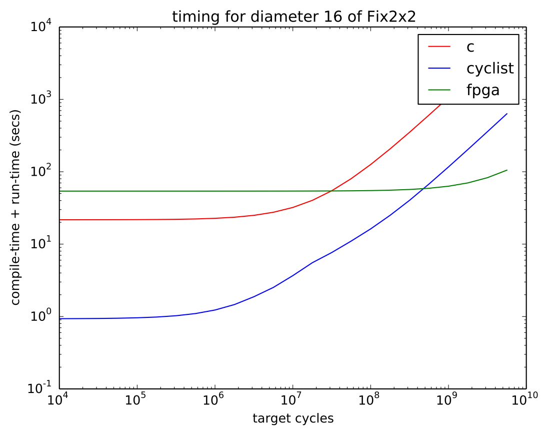


Fig. 15: Total compilation+runtime time over target time steps for emulating a 2x2 Cyclist array for C++, FPGA, and Cyclist emulation systems. Using the runtime performance curve from Figure 13 and "pay as you go" compilation, Cyclist can beat all other emulation systems up to a billion cycles.

in debugging performance is *full visibility*: the ability to track any set of signals, starting at any cycle.

While conceptually simple, full visibility comes with a heavy performance cost. Recording the value of every wire on every cycle requires an enormous amount of bandwidth. For example, on software simulators, generating a full VCD (value change dump) trace results in approximately a 20x slowdown compared to recording only top-level wires. Circuit emulation on FPGAs brings additional complications, requiring design recompilation to adjust visibility of signals to integrated logic analyzers.

To avoid the performance hit of providing full visibility, we introduce the notion of "interactive visibility". During emulation, a snapshot of the state of the circuit is periodically saved. When the user requests the value of a specific wire on a specific cycle, the nearest snapshot is loaded back into the circuit, and the value of the wire is computed by rerunning the circuit until it reaches the desired cycle. The delay between requesting and receiving the value of a wire on a given cycle, is equal to the time needed to emulate the circuit forwards from the nearest snapshot to the desired cycle. We can adjust the frequency at which we take snapshots until the maximum delay is within acceptable interactive latency (e.g. 500 milliseconds). Thus we can *interactively* observe any internal wire, on any cycle, with tolerable latencies.

We have implemented a demonstration debugger, called Scope, that provides interactive visibility. The main screen shows a list of wires and their values across a range of cycles. Users are free to add *any* wire to the list, to scroll backwards and forwards through time, and to search backwards and forwards in time for a given trigger expression to evaluate to true. Scope is essentially a visual interface over two basic commands, view and find.

$$\begin{aligned}&\text{view } (w_1, w_2, \ldots, w_m) \text{ from } n_1 \text{ to } n_2\\&\text{find } e \text{ from } n_1 \text{ to } n_2\end{aligned}$$

View takes as input a list of wires, and a start and end cycle, n_1 and n_2, and returns the values of each wire on every cycle between n_1 and n_2. Find takes a start and end cycle, n_1 and n_2, and returns the first cycle where the trigger expression e evaluates to true, or -1 if e never evaluates to true within n_1 and n_2. Trigger expressions are simple logical and arithmetic expressions based on signal values. Cyclist provides hardware support for these interactive visibility primitives

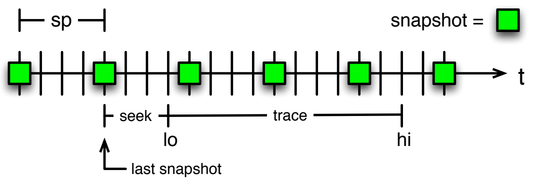


Fig. 16: View command shown over time. To run a view command, we first load the nearest snapshot, run to the first requested cycle, and then run until the last requested cycle with tracing of requested wires enabled.

through fast snapshotting of state and dynamic insertion of trigger circuitry to running circuits. These features are sufficient to provide a low-latency, high throughput experience.

During emulation, Cyclist periodically saves a snapshot of the current circuit state. To implement the view command, we load the nearest snapshot to n_1 into Cyclist, and run the circuit forwards until n_1. We then selectively trace the desired wires from n_1 to n_2. See Figure 16 for a picture of the view command over time. To implement the find command, we first compile the trigger expression to a small combinational circuit and add it to the design under test. We load the nearest snapshot to n_1 into Cyclist, and selectively trace the result of the trigger expression until n_2. The host will scan through the returned results to find the first cycle at which the trigger expression evaluated to true.

Trigger expression compilation runs in time proportional to only the size of the trigger expression. First, the trigger expression nodes are added to the existing positioned graph, and then simulated annealing is applied selectively to the trigger expression nodes resulting in positions for these nodes. From there, the trigger expression nodes are scheduled resulting in placed and scheduled instructions. These instructions are then concatenated to the existing instructions, and this combined code is loaded onto Cyclist. This method can also support sequential trigger expressions that depend on the values of wires across multiple clock cycles by compiling them into *feed-forward* circuits containing registers as well as combinational logic.

In addition to allowing for user-friendly features, the combination of circuit-annotation-based triggers and periodic snapshots requires much less bandwidth than recording the values of every internal wire on every cycle, and we show here that it actually has a negligible performance impact. For a smooth user experience, we require that the maximum delay for the view command to respond be 500 milliseconds. This forces us to take a snapshot for every 500 milliseconds of compute: $rt = 500ms$. The offload time is limited by the size of the snapshot and the bandwidth between Cyclist and the final storage location of the snapshots. With an extremely conservative estimate of 60MB/s off-chip bandwidth and worst-case full utilization of a 1024-tile system with 4KB per tile, there is a performance penalty of only 12% for emulation on Cyclist with periodic snapshots.

Note that interactive visibility depends on the ability to restart from a past cycle. We have thus far assumed that the complete state of the circuit *can* be captured in snapshots, however this is not true of external devices. To provide interactive visibility for circuits with external inputs, we need the ability to rerun the circuit with the *same inputs* observed during the initial run of the circuit. This can be done by recording the inputs received on every clock cycle during the initial run, and then replaying them when needed. Bandwidth requirements can be reduced by compressing the inputs with VCD-style compression.

VI. Discussion and Future Work

Our current Cyclist implementation represents one point in a broad design space of architecture and tools with varying compile and run time performance. Some compilation time improvements could come from parallelizing the layout algorithm, clustering graph nodes by lowering the complexity of the layout, and using a higher radix network to make performance less dependent on layout. There are many options for improving run time performance. The processor could have more pipeline stages to reduce cycle time, instructions could be made wider allowing more work per cycle, and the network could be made flatter to reduce the number of hops between processors. On the ASIC side, better floorplanning and separate clock domains per tile could increase the clock rate. On the compiler front, a better scheduler would reduce the number of instructions, and some form of time space scheduling would potentially produce better run time performance.

Although building an ASIC would give the best raw performance, because of FPGAs great density and economies of scale, constructing a Cyclist FPGA overlay might be an attractive near term option. The idea would be to program an FPGA with a large array of Cyclist tiles and then to use Cyclist tools from then on. Very initial experiments indicate we could achieve 110MHz host clock rate on a mid range Xilinx FPGA.

Cyclist is an emulation system but holds promise for more general computing applications especially those applications that FPGAs are currently used for. A number of changes would allow it to better execute DSP applications for example. Cyclist's debug support, with its snapshotting, tracing, and search, would certainly make it a more productive development platform than what FPGAs currently offer. It could be argued that this debug support would be more generally useful for other multiprocessor architectures or SOCs which are notoriously difficult to debug.

VII. Conclusions

We presented Cyclist, a new cycle-accurate emulation system, that accelerates hardware development. It does this by both decreasing the combined compile + run time and by providing powerful and efficient debugging support including interactive visibility and fast trigger expression compilation. In short, Cyclist is a cost effective emulation design which makes it easy to map designs, has fast compilation and run speeds, and allows visibility at speed without recompilation and in circuit. Cyclist represents a promising direction in reconfigurable computing combining good ideas from both the software and hardware worlds.

VIII. Acknowledgements

Research partially funded by DARPA Award Number HR0011-12-2-0016; the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA; and ASPIRE Lab industrial sponsors and affiliates Intel, Google, HPE, Huawei, LGE, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

References

[1] Q. Inc, "Qualcomm chipsets." http://www.qualcomm.com/, 2014. [Online; accessed August 7, 2014].

[2] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. R. Larus, E. Peterson, G. Prashanth, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceedings of 41st Annual International Symposium on Computer Architecture (ISCA), Minneapolis, MN, June 2014*, IEEE Press, 2014.

[3] S. Inc, "Vcs verilog simulation and verification system." http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/VCS.aspx, 2014. [Online; accessed August 7, 2014].

[4] J. R. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: Constructing Hardware In a Scala Embedded Language," in *Proceedings of 41st Annual International Symposium on Computer Architecture (ISCA), Minneapolis, MN, June 2014*, pp. 1212–1221, IEEE Press, 2014.

[5] M. G. Inc, "Veloce2 emulation system overview." http://www.mentor.com/products/fv/emulation-systems/, 2014. [Online; accessed August 7, 2014].

[6] C. Inc, "Palladium III data sheet." http://www.cadence.com/rl/Resources/datasheets/incisive_enterprise_palladium.pdf, 2014. [Online; accessed August 4, 2014].

[7] P. Wang, J. Collins, C. Weaver, B. Kuttanna, S. Salamian, G. Chinya, E. Schuchman, O. Schilling, S. Steibl, and H. Wang, "Intel Atom processor core made FPGA synthesizable," in *Proceedings of the Seventeenth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'09)*, ACM Press, 2009.

[8] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund, R. Singhal, S. Steibl, and H. Wang, "Intel Nehalem processor core made FPGA synthesizable," in *Proceedings of the Eighteenth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'10)*, ACM Press, 2010.

[9] S. Inc, "Zebu emulation system overview." http://www.synopsys.com/Tools/Verification/hardware-verification/emulation/Pages/default.aspx, 2014. [Online; accessed August 6, 2014].

[10] G. F. Pfister, "The IBM Yorktown Simulation Engine," in *Proceedings of the IEEE*, pp. 850–860, IEEE Press, 1986.

[11] D. K. Beece, G. Deibert, G. Papp, and F. Villante, "The IBM Engineering Verification Engine," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC88)*, pp. 218–224, IEEE Press, 1988.

[12] D. Grant, C. Wang, and G. G. Lemieux, "A CAD framework for Malibu: An FPGA with time-multiplexed coarse-grained elements," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, (New York, NY, USA), pp. 123–132, ACM, 2011.

[13] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The RISC-V instruction set manual, volume I: Base user-level ISA," Tech. Rep. UCB/EECS-2011-62, EECS Department, University of California, Berkeley, May 2011.

[14] C. Wang and G. G. Lemieux, "Scalable and deterministic timing-driven parallel placement for FPGAs," in *Proceedings of the Nineteenth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'11)*, ACM Press, 2011.